REMARKS/ARGUMENTS

Claims 1-10 are pending in the present application. Claims 1-10 are amended to correct antecedent basis issues and typographical errors, both of which are unrelated to the scope and patentability of the claims. Claims 11-20 are new. Support for the amendments to the claims can be found in the claims as originally filed. Support for new claims 11-20 can be found in the specification on page 7, line 9 through page 8, line 2. No new matter has been added. Reconsideration of the claims is respectfully requested.

I. Claim Objections

The examiner objected to claims 1 and 3-5 on the basis that certain terms specified by the examiner lacked antecedent basis. Applicants amended these claims accordingly, thereby overcoming the objections.

II. 35 U.S.C. § 101, Asserted Non-Statutory Subject Matter

The examiner has rejected claim 3 under 35 U.S.C. § 101 as directed towards non-statutory subject matter. Applicants have amended claim 3 to recite a computer program product stored in a computer readable medium. Claim 3 as amended meets the requirements under the MPEP and established case law for statutory subject matter. Therefore, this rejection is overcome.

III. 35 U.S.C. § 103, Asserted Obviousness

III.1. Claims 1, 3, and 4

The examiner rejected claims 1, 3, and 4 under 35 U.S.C. § 103 as obvious over *Sokolov.*, Exception Handling in JAVA Computing Environments, U.S. Patent Application Publication 2003/0079202 (April 24, 2003) (hereinafter "*Sokolov 1*") in view of *Sokolov*, Execution of Synchronized Java Methods in JAVA Computing Environments, U.S. Patent Application Publication 2003/0079203 (April 24, 2003) (hereinafter "*Sokolov 2*"). This rejection is respectfully traversed.

Regarding claim 1, the examiner stated that:

> **Regarding claim 1**, Sokolov(1) discloses a method of handling an exception (abstract, lines 1-2) comprising:
>
> Recognizing that an exception has occurred during the execution of a given method ([0015]. lines 1-5);
>
> Consulting a stack frame associated with said given method ([0024], lines 512).

> Sokolov(l), however, does not teach determining the identity of an object required to be unlocked and removing said lock on said object.
>
> Sokolov(2) does teach determining the identity of an object required to be unlocked ([0015], lines 1-18 disclose the method of executing a synchronized Java method. In the method monitors (locks) are referenced to their association with Java methods and are released based on the appropriate reference that has been determined .) and removing said lock on said object ([L0015], lines 15-17).
>
> Therefore, it would have been obvious to one of ordinary skill in the are at the time of the invention to modify Sokolov(1) by determining the identity of an object required to be unlocked and removing said lock on said object as taught by Sokolov(2), in order to efficiently execute JAVA methods during an exception occurrence.

Office Action of December 19, 2006, pp. 3-4 (emphasis in original).

Claim 1 as amended is as follows:

> 1. A method of handling an exception, the method comprising:
> recognizing that the exception has occurred during execution of a given method;
> consulting a stack frame associated with the given method to determine an identity of an object required to be unlocked; and
> removing a lock on the object.

III.1.i. ***The Examiner Ignored a Feature of Claim 1***

The examiner failed to state a *prima facie* obviousness rejection against claim 1 because the proposed combination of *Sokolov 1* and *Sokolov 2* does not teach or suggest all of the features of claim 1. Specifically, the proposed combination does not teach the claimed feature of, "consulting a stack frame associated with the given method ***to*** determine an identity of an object required to be unlocked."

The examiner does not assert otherwise. In rejecting claim 1, the examiner ignored the feature that the feature of "consulting a stack frame" is "***to*** determine an identity of an object required to be unlocked." Instead, the examiner states that *Sokolov 1* teaches recognizing that an exception has occurred and consulting a stack frame. The examiner then states that *Sokolov 2* teaches "determining." However, in misconstruing the language of claim 1 the examiner has ignored the feature of "consulting a stack frame" "***to*** determine an identity of an object required to be unlocked." Therefore, the rejection, on its face, is not a *prima facie* obviousness rejection with regard to claim 1.

III.1.ii. ***The Proposed Combination Does Not Teach or Suggest All of the Features of Claim 1***

Additionally, the combination of *Sokolov 1* and *Sokolov 2* does not teach or suggest the feature of, "consulting a stack frame associated with the given method ***to*** determine an identity of an object

required to be unlocked," as in claim 1. *Sokolov 1* teaches a method of handling exceptions in a JAVA program, as shown in the following text quoted by the examiner as purportedly showing some of the features of claim 1:

> [0015] As a method for handling exceptions which can be raised during execution of a Java program, one embodiment of the invention includes the acts of: determining that an exception has occurred during the execution of the Java program, and identifying a frame for a Java method. The frame is stored on a Java execution stack when it is determined that an exception has occurred. The Java method is associated with an exception handler suitable for handling said exception. This embodiment can also include the act of accessing a field within the frame, the frame providing a reference to the exception handler which is suitable for handling the exception. This embodiment can also invoke the execution handler to handle the exception by using the reference.
> ...
> [0024] To achieve this and other objects of the invention, improved techniques for handling exceptions raised during the execution of Java computer programs are disclosed. The techniques can be used by a Java virtual machine to efficiently handle exceptions. In one embodiment, a method descriptor is implemented in a Java method frame which is stored in the Java execution stack. The method descriptor provides one or more references to exception handlers associated with the Java method. As will be appreciated, the references can be used to quickly identify and invoke the appropriate exception handler. This can be achieved without having to use a native language execution stack. As a result, the overhead associated with several returns from native functions (routines or methods) can be avoided since the information needed to invoke the appropriate exception handler can be obtained efficiently from the Java execution stack. Accordingly, the performance of Java virtual machines, especially those operating with limited resources, can be significantly enhanced.

Sokolov 1, paragraphs 0015 and 0024.

Specifically, *Sokolov 1* teaches that when an exception occurs in a JAVA method, a particular exception handler is associated with the particular JAVA method. The particular exception handler handles the exception. Still more specifically, the frame provides a reference to the exception handler. Thus, the execution handler can be invoked to handle the exception by using the reference provided by the frame.

As is known in the art, a stack frame is a portion of a call stack. A call stack stores information used during the execution of a program. See, for example, en.wikipedia.org/wiki/Stack_frame, which describes call stacks and stack frames. See also msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/_asug_viewing_the_call_stack_for_a_function.asp for corroboratory information. Thus, *Sokolov 1* teaches that a reference to a particular exception handler can be stored in a frame of a call stack.

Thus, the examiner's characterization that *Sokolov 1* teaches a) recognizing that exception has occurred and, in the most general sense only, b) consulting a stack frame, is appropriate. However,

Sokolov 1 is completely devoid of disclosure regarding, "consulting a stack frame associated with the given method ***to*** determine an identity of an object required to be unlocked," as required by claim 1. Claim 1 specifically requires that consulting the stack frame be performed to determine an identity of an object required to be unlocked. *Sokolov 1* does not teach this specific requirement of claim 1. Because *Sokolov 1* is devoid of disclosure in this regard, *Sokolov 1* also does not suggest this claimed feature.

Additionally, *Sokolov 2* does not teach or suggest the feature of, "consulting a stack frame associated with the given method ***to*** determine an identity of an object required to be unlocked," as required by claim 1. *Sokolov 2* teaches a method of executing synchronized JAVA methods, as shown by the following portion of *Sokolov 2* cited by the examiner as purportedly teaching some of the features of claim 1:

> [0015] As a method for executing a synchronized Java method, one embodiment of the invention includes the acts of: invoking a synchronized Java method, popping a reference to a Java object from a Java execution stack, determining which hash table has a reference to a monitor associated with the Java object, searching a hash table for the monitor associated with the synchronized Java method using an object identifier as a key, the hash table being identified by the searching, the object identifier identifying the Java object, acquiring the monitor associated with the synchronized Java method after the search has been performed, pushing a reference to the monitor on a Java execution stack, executing the synchronized Java method after the monitor has been acquired, popping the reference from the Java execution stack, and releasing the monitor using the reference after the reference has been popped from the Java execution stack, thereby allowing the monitor to be released without performing the searching for the monitor.

Sokolov 2, paragraph 0015.

This portion of *Sokolov 2* teaches the following method. First, a synchronized JAVA method is invoked. Second, a reference to a JAVA object is "popped" from a JAVA execution stack (which Applicants believe is a call stack). Third, the particular hash table that contains a reference to a monitor (or lock) associated with the JAVA object is identified. Fourth, the hash table is searched for the monitor using an object identifier as a key. Fifth, the monitor is acquired. Sixth, a reference to the monitor is "pushed" on the execution stack. *Seventh, the synchronized method is executed after the monitor has been acquired.* Eighth, the monitor is released using the reference, after the reference has been popped from the execution stack. In this manner, the monitor is released without searching for the monitor.

Stated differently, *Sokolov 2* teaches that a lock can be released without searching for the lock. *Sokolov 2*'s method is to first acquire the lock ***from a hash table*** and then store the lock in the call stack. Thus, the lock can be easily retrieved and released. However, the method presented in *Sokolov 2 requires*

first finding the lock in the JAVA method by popping the object from the call stack and then finding a reference to the lock ***in the hash table****. Sokolov 2 also requires that the JAVA method be executed after the monitor has been acquired.*

Thus, the method of claim 1 contains three marked, fundamental differences over the method shown in *Sokolov 2*. First, the method of claim 1 consults *a stack frame* to determine an identity of an object required to be unlocked, whereas the method of *Sokolov 2* consults a *hash table* to determine that identity. The hash table in *Sokolov 2* is not part of the call stack, which is why *Sokolov 2* must first "pop" the JAVA object from the execution stack. Thus, *Sokolov 2* does not teach consulting *a stack frame* to determine the identity of the object required to be unlocked.

Second, *Sokolov 2* does not teach consulting the stack frame *to determine* the identity of the object required to be unlocked, as in claim 1. As shown above, *Sokolov 2* consults the hash table, which is distinct from the call stack, to determine the identity of a monitor (or lock).

Third, *Sokolov 2* requires that the JAVA method is executed ***after*** the monitor (lock) has been acquired. Thus, again, the *stack frame* associated with the given method is not consulted *to determine* an identity of an object required to be unlocked, as in claim 1.

Thus, *Sokolov 2* does not teach the claimed feature of, "consulting a stack frame associated with the given method to determine an identity of an object required to be unlocked," as in claim 1. Given the marked differences between the method shown in *Sokolov 2* and claim 1, *Sokolov 2* also does not suggest this claimed feature.

As shown above, *Sokolov 1* does not teach or suggest this claimed feature. Because neither *Sokolov 1* nor *Sokolov 2* teach or suggest this claimed feature, the proposed combination of these references when considered as a whole also does not teach or suggest this claimed feature. Accordingly, the examiner failed to state a *prima facie* obviousness rejection against claim 1.

Claims 3 and 4 contain features similar to those presented in claim 1. Therefore, the examiner failed to state a *prima facie* obviousness rejection against these claims at least for the reasons presented above. Accordingly, the rejection of claims 1, 3, and 4 has been overcome.

III.1.iii. *No Teaching, Suggestion, or Motivation Exists to Combine the References Because the References Are Directed Toward Different Problems*

Additionally, the examiner failed to state a *prima facie* obviousness rejection because no teaching, suggestion, or motivation exists to combine the references in the manner proposed by the examiner. No teaching, suggestion, or motivation exists because the references are directed towards solving different problems.

It is necessary to consider the reality of the circumstances--in other words, common sense--in deciding in which fields a person of ordinary skill would reasonably be expected to look for a solution to the problem facing the inventor. *In re Oetiker*, 977 F.2d 1443 (Fed. Cir. 1992); *In re Wood*, 599 F.2d 1032, 1036, 202 U.S.P.Q. 171, 174 (CCPA 1979). In the case at hand, the cited references address distinct problems. Thus, no common sense reason exists to establish that one of ordinary skill would reasonably be expected to look for a solution to the problem facing the inventor. Accordingly, no teaching, suggestion, or motivation exists to combine the references and the examiner has failed to state a *prima facie* obviousness rejection of claim 1.

For example, *Sokolov 1* is directed to solving the problem of high computing overhead cost of handling exceptions in JAVA programs. For example, *Sokolov 1* provides that:

> [0009] Sometimes during the execution of a Java method, an exception is raised (e.g., divide by zero). This situation typically requires invocation of an exception handler. One problem with the conventional approaches to exception handling in Java computing environments is that there is a significant overhead associated with invoking the appropriate exception handler when an exception is raised. This is partly attributed to the fact that the exception handler can be associated with a Java method that is several levels deep (i.e., exception has occurred during execution of a Java method which has been invoked by a Java method that has been invoked by another Java method, and so on).
>
> [0010] Moreover, conventional approaches can require several returns to be made from native functions (procedures or subroutines) written in a non-Java programming language (e.g., C or C++) in order to identify the appropriate exception handler. This can significantly hinder the performance of Java virtual machines, especially those operating with limited memory and/or limited computing power (e.g., embedded systems).

Sokolov 1, paragraphs 0009 and 0010.

On the other hand, *Sokolov 2* is directed to the problem of high overhead computing costs required by the execution of some synchronized JAVA programs. For example, *Sokolov 2* provides as follows:

> [0009] One problem with conventional techniques for executing synchronized Java methods is that several operations have to be performed in order to execute the synchronized Java methods. Moreover, these operations are performed once when the monitor is acquired and then have to be repeated again in order to release the monitor. Accordingly, it is highly desirable to reduce the overhead associated with execution of synchronized Java methods. This, in turn, can improve the performance of virtual machines, especially those operating with limited resources.

Sokolov 2, paragraph 0009.

Based on the plain disclosures of the references themselves, the references address completely distinct problems that are unrelated to each other. The problem of high computing overhead cost of

handling exceptions in JAVA programs is completely distinct from the problem of high overhead computing costs required by the execution of some synchronized JAVA programs. Although both *Sokolov 1* and *Sokolov 2* are directed towards reducing the overhead computing cost of JAVA programs, the two references are directed to completely different aspects of the JAVA programs. *Sokolov 1* is directed towards exception handling, whereas *Sokolov 2* is directed towards execution of synchronized JAVA programs. One of ordinary skill would have no reason to combine the references because one of ordinary skill would believe that each reference completely solves the problem faced by the corresponding reference.

Because the references address completely distinct problems, one of ordinary skill would have no reason to combine or otherwise modify the references to achieve the invention of claim 1. Thus, no proper teaching, suggestion, or motivation exists to combine the references in the manner suggested by the examiner. Accordingly, the examiner has failed to state a *prima facie* obviousness rejection against claims 1, 3, and 4.

III.2. Claims 5-10

The examiner rejected claims 5-10 under 35 U.S.C. § 103(a) as obvious over *Bak et al.*, Method and Apparatus for Concurrent Thread Synchronization, U.S. Patent 6,167,424 (December 26, 2000) (hereinafter "*Bak*") in view of *Cohn et al.*, Software Mechanism for Reducing Exceptions Generated by Speculatively Scheduled Instructions, U.S. Patent 5,901,308 (May 4, 1999) (hereinafter "*Cohn*"). This rejection is respectfully traversed. Regarding claim 5, the examiner stated that:

> **Regarding claim 5**, Bak discloses at a just in time compiler of programming language code, said programming language code including a plurality of instructions, a method of generating executable code (column 24, lines 61 -66) comprising:
>
> Determining a synchronization depth for said each instruction (column 3, lines 14-29 and column 4, lines 14-25 disclose the method of which thread synchronization is determined based on maintaining a counter used for determining the amount of times a thread has locked and unlocked an object.);
>
> Bak does not disclose that the method comprises:
>
> Associating said synchronization depth with a program counter address associated with said each instruction;
>
> Determining a continuous range of program counter addresses associated with an equivalent synchronization depth; and
>
> Storing an indication of said continuous range of program counter addresses in a table associated with said synchronization depth.

However, Cohn discloses that the method comprises:

Associating said synchronization depth with a program counter address associated with said each instruction (column 7, lines 18-20 and column 7, lines 37-40);

Determining a continuous range of program counter addresses associated with an equivalent synchronization depth (column 7, lines 37-45); and

Storing an indication of said continuous range of program counter addresses in a table associated with said synchronization depth (Column 3, lines 28-41 disclose a table used to store a range of address locations identifying instructions causing exceptions.).

Therefore, it would have been obvious to one of ordinary skill in the art at the time of the invention to modify Bak by associating said synchronization depth with a program counter address, determining a continuous range of program counter addresses associated with an equivalent synchronization depth, and storing an indication of said continuous range of program counter addresses in a table as taught by Cohn, for the benefit of efficiently executing instructions during the occurrence of an exception.

Office Action of December 19, 2006, pp. 6-7 (emphasis in original).

Claim 5 as amended is as follows:

5. (Currently Amended) A method of generating executable code at a just in time compiler of programming language code, wherein the programming language code includes a plurality of instructions, the method comprising:
determining a synchronization depth for each instruction in the plurality of instructions;
associating the synchronization depth with a program counter address associated with each instruction in the plurality of instructions;
determining a continuous range of program counter addresses associated with an equivalent synchronization depth; and
storing an indication of the continuous range of program counter addresses in a table associated with the equivalent synchronization depth.

III.2.i. ***The Proposed Combination Does Not Teach or Suggest All of the Features of Claim 5***

The examiner failed to state a *prima facie* obviousness rejection against claim 5 because the proposed combination, when considered as a whole, does not teach or suggest most of the claimed features. For example, the proposed combination of *Cohn* and *Bak* does not teach or suggest the claimed feature of, "determining a synchronization depth for each instruction in the plurality of instructions," as in claim 1. The examiner asserts otherwise, citing the following portion of *Bak*:

When thread 202 attempts to execute a synchronized operation on object 204, a synchronization construct 206 which is associated with object 204 is obtained. In general, object 204 is dynamically associated with a synchronization construct, as for example synchronization construct 206a, which is arranged to

> provide synchronized access to object 204. If synchronization construct 206a permits re-entrant locking of object 204, it may include a counter 208 which may be incremented to keep track of the number of times object 204 has been locked by thread 202. Synchronization construct 206a further includes an object pointer 210 that identifies object 204 or, more generally, the object with which monitor 206a is associated, Synchronization construct 206a also includes an identifier for thread 202, the thread that currently has locked synchronization construct 206a.

Bak, col. 3, ll. 14-29.

This portion of *Bak* teaches that when a thread attempts to execute a synchronized operation on an object, a synchronization construct is obtained. A counter is used to count the number of times an object has been locked during re-entrant locking of the object. The synchronization construct has other features, such as a pointer to identify the object and an identifier for the thread that locked a construct associated with the object.

However, nothing in this portion of *Bak* teaches determining a synchronization depth for each instruction in the plurality of instructions, as in claim 5. Although *Bak* counts the number of times an object is locked for a thread, this feature does not teach or suggest determining a synchronization depth *for each instruction* in the plurality of instructions. For example, a thread usually has multiple instructions (a plurality of instructions is claimed). In another example, a plurality of instructions, as claimed, is not necessarily a plurality of threads and any given instruction may be part of the same thread as another instruction. However, *Bak* does not keep track of the number of times *each instruction* causes an object to be locked; instead, *Bak* keeps track of the number of times the overall thread causes an object to be locked. Thus, even if the feature of counting locks caused by a thread, as in *Bak*, is equivalent to determining a synchronization depth, as claimed (which Applicants dispute), *Bak* does not satisfy the claimed requirement of, "for each instruction in the plurality of instructions." In other words, because *Bak* teaches counting the number of locks caused by a thread and because a thread is different than a plurality of instructions, as claimed, *Bak* does not teach or suggest, "determining a synchronization depth for each instruction in the plurality of instructions," as in claim 5.

Nevertheless, the examiner also refers to the following portion of *Bak* as teaching this claimed feature:

> If synchronization constructs are to support re-entrant locking, they may also require explicit counters which are used to track the number of times a given thread relocks an object that it has already locked. The implementation and maintenance of explicit counters may be relatively expensive in terms of the use of computer system resources. Further, since the synchronization construct explicitly keeps track of the thread that has locked it, the synchronization

> construct must be continually updated. Continually updating the synchronization construct is typically both time-consuming and expensive in terms of the consumption of computer system resources.

Bak, col. 4, ll. 14-25.

Again, *Bak* describes tracking the number of times a given thread relocks an object. *Bak* also describes the problem with the resulting need to continually update a synchronization object. However, as with the above quote, *Bak* does not teach or suggest, "determining a synchronization depth for each instruction in the plurality of instructions," as in claim 5, because counting locks created by a thread is different than determining a synchronization depth for each instruction in a plurality of instructions. Thus, again, *Bak* does not teach or suggest this claimed feature. Furthermore, nothing else in *Bak* teaches or suggests this claimed feature.

Additionally, *Cohn* does not teach or suggest this claimed feature. *Cohn* is directed to a method for reducing the number of speculative exceptions. *Cohn* teaches flagging certain speculative instructions likely to cause exceptions and, during compilation, avoiding execution of those certain speculative instructions. *Cohn*, Abstract. Additionally, as shown below, nothing in *Cohn* teaches or suggests "determining a synchronization depth for each instruction in the plurality of instructions," as in claim 5.

Because neither *Bak* nor *Cohn* teach or suggest this claimed feature, the proposed combination of these references, when considered as a whole, also does not teach or suggest this claimed feature. Accordingly, the examiner failed to state a *prima facie* obviousness rejection against claim 5.

Additionally, the combination of *Bak* and *Cohn*, considered as a whole, does not teach or suggest, "associating the synchronization depth with a program counter address associated with each instruction in the plurality of instructions," as in claim 5. The examiner asserts otherwise, citing the following portion of *Cohn*:

> Once the instructions likely to cause exceptions have been detected, the method, according to this invention, reduces the occurrence of speculative exceptions by identifying those instructions most likely to cause exceptions, and precluding the re-scheduling of those instructions. Referring again to FIG. 1, upon completion of the initial compilation process, the compiler 16 outputs an executable file 25, representing the optimized application in scheduled, instruction set format. In addition, the compiler outputs a translation table 20. The translation table 20 includes an entry for each instruction. An example of a translation table entry 30 is shown in FIG. 2A to include a program counter 31 and a tag 32. *As mentioned previously, the program counter indicates the location, in memory of the instruction in the executable program.* The tag is a pointer to where the instruction ended up in the internal representation of the compiled program. Alternatively, the tag could be the location of the instruction in the source code.
>
> ...

> *During the profiling phase, the program counters of each of the instructions that cause exceptions are recorded in a PC log file 23, and a count of the exceptions associated with each program counter is maintained. The PC log file 23 is actually a data structure stored in memory, where each entry of the data structure includes a program counter and a count. Each time an exception occurs, the PC log file 23 is searched for the appropriate program counter, and the counter corresponding to the program counter is updated.* A hash table is maintained to locate the appropriate program counter in the PC log file, although other methods, known to those of skill in the art, may also be used. A separate PC log is generated for each data set. When all the representative data sets have been executed, the PC logs for each data set are merged to provide one large data structure PC Log 23 including the program counters that caused the exceptions during the profiling run, and the corresponding number of exceptions caused by each PC.

Cohn, col. 7, ll. 6-23 and ll. 37-54 (emphasis to show portions cited by the examiner).

This portion of *Cohn* teaches reducing speculative exceptions by precluding re-scheduling of instructions likely to cause such exceptions. A program counter is provided for each instruction that causes exceptions. The counter *counts the number of exceptions associated with each program*. The program counter also indicates the location, in memory, of each instruction that causes an exception.

However *Cohn* does not teach associating a *synchronization depth* with a program counter depth, as in claim 5. Assuming, *arguendo*, that the program counter indication of an instruction memory location in *Cohn* is equivalent to the claimed program counter address (an assertion Applicants dispute), *Cohn* still does not describe associating a synchronization depth with the program counter indication of an instruction memory location. In fact, *Cohn* is completely devoid of disclosure regarding synchronization – as the examiner can confirm via a simple keyword search of *Cohn*.

For similar reasons, *Cohn* does not teach or suggest, "determining a continuous range of program counter addresses *associated with an equivalent synchronization depth*," or "storing an indication of the continuous range of program counter addresses in a table *associated with the equivalent synchronization depth*," as in claim 5. In particular, because *Cohn* is devoid of disclosure regarding synchronization, *Cohn* does not teach or suggest determining a continuous range of program counter addresses *associated with* an equivalent synchronization depth. Additionally, *Cohn* does not teach or suggest storing an indication of the continuous range of program counter addresses in a table *associated with* the equivalent synchronization table.

The examiner admits, and Applicants agree, that *Bak* does not teach these claimed features. By implication, the examiner also appears to agree that *Bak* does not suggest these claimed features. Additionally, nothing in *Bak* actually teaches or suggests these claimed features.

Because neither *Cohn* nor *Bak* teach or suggest the features of claim 5, as asserted by the examiner, the proposed combination of these references, when considered as a whole, also does not teach

or suggest these claimed features. Therefore, the examiner failed to state a *prima facie* obviousness rejection against claim 5. For similar reasons, the examiner failed to state a *prima facie* obviousness rejection against claims 6-10.

III.2.ii. *No Teaching, Suggestion, or Motivation Exists to Combine the References Because the References Are Directed Toward Different Problems*

Additionally, the examiner failed to state a *prima facie* obviousness rejection because no teaching, suggestion, or motivation exists to combine the references in the manner proposed by the examiner. No teaching, suggestion, or motivation exists because the references are directed towards solving different problems.

For example, *Bak* is directed to solving the problem of high computing overhead cost of handling exceptions in synchronized programs. For example, *Bak* provides that:

> The use of monitors as synchronization constructs to track the status of objects is often relatively inefficient in that a software cache or a hash table of synchronization constructs must typically be searched in order to locate the proper monitor for use with a given object. Such searches may prove to be time-consuming, and generally utilize relatively large amounts of computer system resources. The cache of synchronization constructs, in itself, typically occupies a significant amount of computer memory. In addition, the memory management associated with allocating a monitor for an object when a suitable monitor does not already exist may be costly. Finally, as synchronization construct caches may be shared among multiple threads, they themselves may have to be locked prior to access or update, which both imposes additional costs in execution time and also introduces a source of locking contention that occurs when more than one thread wants to access the synchronization construct cache at one time.

Bak, col. 3, l. 63 through col. 4, l. 13.

On the other hand, *Cohn* is directed to the problem of reducing the number of speculative exceptions in a program. For example, *Cohn* provides as follows:

> Thus both hardware and software techniques have been provided for correcting problems associated with speculative exceptions. However, each technique brings with it an associated overhead. The hardware technique utilizes chip area for control and pipelining of data. The software technique requires compute cycles for determining whether the exception handler need, in fact be executed, thus reducing the performance of the computer system. It is desirable to reduce the number of speculative exceptions to thereby minimize the overhead incurred by the software and hardware exception handling techniques.

Cohn, col. 3, ll. 10-20.

Based on the plain disclosures of the references themselves, the references address completely distinct problems that are unrelated to each other. The problem of high computing overhead cost of handling exceptions in synchronized programs is completely distinct from the problem of reducing the

number of speculative exceptions in a program. Although both *Bak* and *Cohn* are directed towards reducing the overhead computing cost of programs, the two references are directed to completely different aspects of the reducing overhead in such programs. *Bak* is directed towards exception handling, whereas *Cohn* is directed towards reducing speculative exceptions. One of ordinary skill would have no reason to combine the references because one of ordinary skill would believe that each reference completely solves the problem faced by the corresponding reference.

Because the references address completely distinct problems, one of ordinary skill would have no reason to combine or otherwise modify the references to achieve the invention of claim 5. Thus, no proper teaching, suggestion, or motivation exists to combine the references in the manner suggested by the examiner. Accordingly, the examiner has failed to state a *prima facie* obviousness rejection against claim 5 or against related claims 6-10.

III.3. Claim 2

The examiner rejected claim 2 under 35 U.S.C. § 103(a) as obvious over *Sokolov 1*, *Sokolov 2*, and *Yoshioka et al.*, Accessing Exception Handlers without Translating the Address, U.S. Patent 6,425,039 (July 23, 2002) (hereinafter "*Yoshioka*"). This rejection is respectfully traversed. The examiner states that:

> **Regarding claim 2**, Sokolov(1) and Sokolov(2) do not disclose determining a program counter address at which said exception has occurred;
>
> Determining, from a table. a synchronization depth associated with said program counter address; and
>
> Wherein said consulting said stack frame includes reading from a location associated with said synchronization depth.
>
> However, Yoshioka discloses determining a program counter address at which said exception has occurred (column 3, lines 13-18);
>
> Determining, from a table, a synchronization depth associated with said program counter address (column 14, lines 4-13); and,
>
> Wherein said consulting said stack frame includes reading from a location associated with said synchronization depth (column 2, lines 12-18).
>
> Therefore, it would have been obvious to one of ordinary skill in the are at the time of the invention to modify Sokolov(1) and Sokolov(2) by, determining a program counter address at which an exception has occurred, Determining, from a table, a synchronization depth associated with the program counter address, and

> when consulting the stack frame reading from a location associated with the synchronization depth as taught by Yoshioka, for the benefit of reducing execution time when handling exceptions.

Office Action of December 19, 2006, p. 12 (emphasis in original).

The examiner failed to state a *prima facie* obviousness rejection against claim 2 because the proposed combination does not teach all of the features of claim 2. As shown above, the proposed combination of *Sokolov 1* and *Sokolov 2* does not teach or suggest the features of claim 1, as asserted by the examiner. *Yoshioka* fails to cure the lack of disclosure in *Sokolov 1* and *Sokolov 2* in this regard. Thus, the proposed combination of these references, considered as a whole, also fails to teach or suggest all of the features of claim 2. Accordingly, the examiner failed to state a *prima facie* obviousness rejection against claim 2.

Instead, *Yoshioka* is directed towards the problem reducing the time between the occurrence of an exception and the handling of an exception. For example, *Yoshioka* provides as follows:

> The present invention relates to a data processor for executing an exception handling program to cope with the occurrence of exceptions such as reset event, exception events and interrupt events. More specifically, the invention relates to technology for shortening the time required for the transition from a moment of occurrence of an exception event to the operation of an exception handler for coping with the exception event. The invention relates to technology that can be effectively adapted to, for example, a single-chip microcomputer or a microprocessor contained in a memory management unit (MMU).

Yoshioka, col. 1, ll. 8-17.

This problem is completely distinct from the problem of high computing overhead cost of handling exceptions in JAVA programs, as in *Sokolov 1*. This problem is also completely distinct from the problem of high overhead computing costs required by the execution of some synchronized JAVA programs, as in *Sokolov 2*. As shown above, the problems addressed by *Sokolov 1* and *Sokolov 2* are also completely distinct. Given that none of the problems addressed by the references are related to each other, no one of ordinary skill would be motivated to combine the references. Thus, no such teaching, suggestion, or motivation exists contrary to the examiner's assertion. Accordingly, the examiner failed to state a *prima facie* obviousness rejection against claim 2.

IV. New Claims 11-20

New claims 11-20 contain features also not taught or suggested by the cited references or by other known references. For example, none of the references teach or suggest consulting *only* the stack frame, as in claim 11. None of the references teach or suggest executing a syncenter bytecode or a syncexit

bytecode as in claims 13 and 15 respectively. Similarly, none of the references teach or suggest the remaining claims in new claims 11-20. Therefore, no *prima facie* obviousness rejection can be made against these new claims using known references.

V. Conclusion

The subject application is patentable over the cited references and should now be in condition for allowance. The examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: March 19, 2007

Respectfully submitted,

/Theodore D. Fay III/

Theodore D. Fay III
Reg. No. 48,504
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 385-8777
Attorney for Applicants